Understanding and Using Execution Context in Windows NT Drivers

[ Classic Readings at ADVDBG.ORG ]

Open Systems Resources, Inc.
The NT Insider Vol.03, Issue 02
April 1996
&&
Microsoft Windows Driver Development Kit
Microsoft Windows Server 2003 Service Pack 1 (SP1) and Earlier
Built on Friday, February 18, 2005
Contents

1 Introduction 2
2 What is Context? 4
3 Different Contexts 6
4 Impacts 9
5 Take It To The Limit 12
6 Summary 15
Chapter 1

Introduction

One of the most important concepts to understand about drivers in Windows NT is the "execution context" in which the drivers run. Understanding this concept, and applying what you know about it carefully, can help you build faster, more efficient drivers.

This article includes the following topics:

- What is Context?
- Different Contexts
- Impacts
- Take It To The Limit
- Summary

An important aspect of NT standard kernel mode drivers is the "context" in which particular driver functions execute. Traditionally of concern mostly to file systems developers, writers of all types of NT kernel mode drivers can benefit from a solid understanding of execution context. When used with care, understanding execution context can enable the creation of higher performance, lower overhead device driver designs.

In this article, we’ll explore the concept of execution context. As a demonstration of the concepts presented, this article ends with the description of a
driver that allows user applications to execute in kernel mode, with all the rights and privileges thereof. Along the way, we’ll also discuss the practical uses that can be made of execution context within device drivers.
Chapter 2

What is Context?

When we refer to a routine’s context we are referring to its thread and process execution environment. In NT, this environment is established by the current Thread Environment Block (TEB) and Processes Environment Block (PEB). Context therefore includes the virtual memory settings (telling us which physical memory pages correspond to which virtual memory addresses), handle translations (since handles are process specific), dispatcher information, stacks, and general purpose and floating point register sets. When we ask in what context a particular kernel routine is running, we are really asking, ”What’s the current thread as established by the (NT kernel) dispatcher?” Since every thread belongs to only one process, the current thread implies a specific current process. Together, the current thread and current process imply all those things (handles, virtual memory, scheduler state, registers) that make the thread and process unique.

Virtual memory context is perhaps the aspect of context that is most useful to kernel mode driver writers. Recall that NT maps user processes into the low 2GB of virtual address space, and the operating system code itself into the high 2GB of virtual address space. When a thread from a user process is executing, it’s virtual addresses will range from 0 to 2GB, and all addresses above 2GB will be set to ”no access,” preventing direct user access to operating system code and structures. When the operating system code is executing, its virtual addresses range from 2-4GB, and the current user process (if there is one) is mapped into the addresses between 0 and 2GB.
In NT V3.51 and V4.0, the code mapped into the high 2GB of address never changes. However, the code mapped into the lower 2GB of address space changes, based on which process is current.

In addition to the above, in NT’s specific arrangement of virtual memory, a given valid user virtual address X within process P (where X is less than or equal to 2GB) will correspond to the same physical memory location as kernel virtual address X. This is true, of course, only when process P is the current process and (therefore) process P’s physical pages are mapped into the operating system’s low 2GB of virtual addresses. Another way of expressing this last sentence is, ”This is true only when P is the current process context.” So, user virtual addresses and kernel virtual addresses up to 2GB refer to the same physical locations, given the same process context.

Another aspect of context of interest to kernel mode driver writers is thread scheduling context. When a thread waits (such as by issuing the Win32 function WaitForSingleObject() for an object that is not signaled), that thread’s scheduling context is used to store information which defines what the thread is waiting for. When issuing an unsatisfied wait, the thread is removed from the ready queue, to return only when the wait has been satisfied (by the indicated dispatcher object being signaled).

Context also impacts the use of handles. Since handles are specific to a particular process, a handle created within the context of one process will be of no use in another processes context.
Chapter 3

Different Contexts

Kernel mode routines run in one of three different classes of context:

- *System* process context
- A specific user thread (and process) context
- Arbitrary user thread (and process) context

During its execution, parts of every kernel mode driver might run in each of the three context classes above. For example, a driver’s `DriverEntry()` function always runs in the context of the system process. System process context has no associated user-thread context (and hence no TEB), and also has no user process mapped into the lower 2GB of the kernel’s virtual address space. On the other hand, DPCs (such as a driver’s DPC for ISR or timer expiration function) run in an arbitrary user thread context. This means that during the execution of a DPC, any user thread might be the ”current” thread, and hence any user process might be mapped into the lower 2GB of kernel virtual addresses.

The context in which a driver’s dispatch routines run can be particularly interesting. In many cases, a kernel mode driver’s dispatch routines will run in the context of the calling user thread. Figure 3.1 shows why this is so. When a user thread issues an I/O function call to a device, for example by calling the Win32 `ReadFile()` function, this results in a system service request. On
Intel architecture processors, such requests are implemented using software interrupts which pass through an interrupt gate. The interrupt gate changes the processor’s current privilege level to kernel mode, causes a switch to the kernel stack, and then calls the system service dispatcher. The system service dispatcher in turn, calls the function within the operating system that handles the specific system service that was requested. For `ReadFile()` this is the `NtReadFile()` function within the I/O Subsystem. The `NtReadFile()` function builds an IRP, and calls the read dispatch routine of the driver that corresponds to the file object referenced by the file handle in the `ReadFile()` request. All this happens at IRQL PASSIVE_LEVEL.

![Diagram](image)

Figure 3.1: Following the Context of the Calling User’s Thread

Throughout the entire process described above, no scheduling or queuing of the user request has taken place. Therefore, no change in user thread and process context could have taken place. In this example, then, the driver’s dispatch routine is running in the context of the user thread that issued the `ReadFile()` request. This means that when the driver’s read dispatch function is running, it is the user thread executing the kernel mode driver code.
Does a driver’s dispatch routine always run in the context of the requesting user thread? Well, no. Section 16.4.1.1 of the V4.0 *Kernel Mode Drivers Design Guide* tells us, ”Only highest-level NT drivers, such as File System Drivers, can be sure their dispatch routines will be called in the context of such a user-mode thread.” As can be seen from our example, this is not precisely correct. It is certainly true that FSDs will be called in the context of the requesting user thread. The fact is that any driver called directly as a result of a user I/O request, without first passing through another driver, is guaranteed to be called in the context of the requesting user thread. This includes FSDs. But it also means that most user-written standard kernel mode drivers providing functions directly to use applications, such as those for process control devices, will have their dispatch functions called in the context of the requesting user thread.

In fact, the only way a driver’s dispatch routine will not be called in the context of the calling user’s thread is if the user’s request is first directed to a higher level driver, such as a file system driver. If the higher level driver passes the request to a system worker thread, there will be a resulting change in context. When the IRP is finally passed down to the lower level driver, there is no guarantee that the context in which the higher level driver was running when it forwarded the IRP was that of the original requesting user thread. The lower-level driver will then be running in an arbitrary thread context.

The general rule, then, is that when a device is accessed directly by the user, with no other drivers intervening, the driver’s dispatch routines for that device will always run in the context of the requesting user thread. As it happens, this has some pretty interesting consequences, and allows us to do some equally interesting things.
Chapter 4

Impacts

What are the consequences of dispatch functions running in the context of the calling user thread? Well, some are useful and some are annoying. For example, let’s suppose a driver creates a file using the ZwCreateFile() function from a dispatch function. When that same driver tries to read from that file using ZwReadFile(), the read will fail unless issued in the context of the same user process from which the create was issued. This is because both handles and file objects are stored on a per-process basis.

Continuing the example, if the ZwReadFile() request is successfully issued, the driver could optionally choose to wait for the read to be completed by waiting on an event associated with the read operation. What happens when this wait is issued? The current user thread is placed in a wait state, referencing the indicated event object. So much for asynchronous I/O requests! The dispatcher finds the next highest priority runnable thread. When the event object is set to signaled state as a result of the ReadFile() request completing, the driver will only run when the user’s thread is once again one of the N highest priority runnable threads on an N CPU system.

Running in the context of the requesting user thread can also have some very useful consequences. For example, calls to ZwSetInformationThread() using a handle value of -2 (meaning ”current thread”) will allow the driver to change all of the current thread’s various properties. Similarly, calls ZwSetInformationProcess() using a handle value of NtCurrentProcess(), which ntddk.h defines as ”-1” will allow the driver to change the characteristics of the
current process. Note that since both of these calls are issued from kernel mode, no security checks are made. Thus, it is possible this way to change thread and/or process attributes that the thread itself could not access.

However, it is the ability to directly access user virtual addresses that is perhaps the most useful consequence of running within the context of the requesting user thread. Consider, for example, a driver for a simple shared-memory type device that is used directly by a user-mode application. Let’s say that a write operation on this device comprises copying up to 1K of data from a user’s buffer directly to a shared memory area on the device, and that the devices shared memory area is always accessible.

The traditional design of a driver for this device would probably use **Buffered I/O** since the amount of data to be moved is significantly less than a page in length. Thus, on each write request the I/O Manager will allocate a buffer in non-paged pool that is the same size as the user’s data buffer, and copy the data from the user’s buffer into this non-page pool buffer. The I/O Manager will then call the driver’s write dispatch routine, providing a pointer to the non-paged pool buffer in the IRP (in `Irp->AssociatedIrp.SystemBuffer`). The driver will then copy the data from the non-paged pool buffer to the device’s shared memory area. How efficient is this design? Well, for one thing the data is always copied twice. Not to mention the fact that the I/O Manager needs to do the pool allocation for the nonpaged pool buffer. I would not call this the lowest possible overhead design.

Say we try to increase the performance of this design, again using traditional methods. We might change the driver to use **Direct I/O**. In this case, the page containing the user’s data will be probed for accessibility and locked in memory by the I/O Manager. The I/O Manager will then describe the user’s data buffer using a **Memory Descriptor List** (MDL), a pointer to which is supplied to the driver in the IRP (at `Irp->MdlAddress`). Now, when the driver’s write dispatch function gets the IRP, it needs to use the MDL to build a system address that it can use as a source for its copy operation. This entails calling `IoGetSystemAddressForMdl()`, which in turn calls `MmMapLockedPages()` to map the page table entries in the MDL into kernel virtual address space. With the kernel virtual address returned from `IoGet-`
SystemAddressForMdl(), the driver can then copy the data from the user’s buffer to the device’s shared memory area. How efficient is this design? Well, it’s better than the first design. But mapping is also not a low overhead operation.

So what’s the alternative to these two conventional designs? Well, assuming the user application talks directly to this driver, we know that the driver’s dispatch routines will always be called in the context of the requesting user thread. As a result we can bypass the overhead of both the direct and buffered I/O designs by using "Neither I/O." The driver indicates that it wants to use "neither I/O" by setting neither the DO_DIRECT_IO nor the DO_BUFFERED_IO bits in the flags word of the device object. When the driver’s write dispatch function is called, the user-mode virtual address of the user’s data buffer will be located in the IRP at location Irp->UserBuffer. Since kernel mode virtual addresses for user space locations are identical to user-mode virtual addresses for those same locations, the driver can use the address from Irp->UserBuffer directly, and copy the data from the user data buffer to the device’s shared memory area. Of course, to prevent problems with user buffer access the driver will want to perform the copy within a try...except block. No mapping; No recopy; No pool allocations. Just a straight-forward copy. Now that’s what I’d call low overhead.

There is one down-side to using the "neither I/O" approach however. What happens if the user passes a buffer pointer that is valid to the driver, but invalid within the user’s process? The try...except block won’t catch this problem. One example of such a pointer might be one that references memory that is mapped read-only by the user’s process, but is read/write from kernel mode. In this case, the move of the driver will simply put the data in the space that the user application sees as read-only! Is this a problem? Well, it depends on the driver and the application. Only you can decide if the potential risks are worth the rewards of this design.

1 Neither I/O Input buffer : pIrpStack–>Parameters.DeviceIoControl.Type3InputBuffer
Chapter 5

Take It To The Limit

A final example will demonstrate many of the possibilities of a driver running within the context of the requesting user thread. This example will demonstrate that when a driver is running, all that’s happening is that the driver is running in the context of the calling user process in kernel mode.

We have written a pseudo-device driver called SwitchStack. Since it is a pseudo-device driver it is not associated with any hardware. This driver supports create, close, and a single IOCTL using METHOD_NEITHER. When a user application issues the IOCTL, it provides a pointer to a variable of type void as the IOCTL input buffer and a pointer to a function (taking a pointer to void and returning void) as the IOCTL output buffer. When processing the IOCTL, the driver calls the indicated user function, passing the PVOID as a context argument. The resulting function, within the user’s address space, then executes in kernel mode.

Given the design of NT, there is very little that the called-back user function cannot do. It can issue Win32 function calls, pop up dialog boxes, and perform File I/O. The only difference is that the user-application is running in kernel mode, on the kernel stack. When an application is running in kernel mode it is not subject to privilege limits, quotas, or protection checking. Since all functions executing in kernel mode have IOPL, the user application can even issue IN and OUT instructions (on an Intel architecture system, of course). Your imagination (coupled with common sense) is the only limit on the types of things you could do with this driver.
NTSTATUS SwitchStackDispatchIoctl(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
    PIO_STACK_LOCATION Ios;
    NTSTATUS Status;
    // Get a pointer to current I/O Stack Location
    Ios = IoGetCurrentIrpStackLocation(Irp);
    // Make sure this is a valid IOCTL for us...
    if(Ios->Parameters.DeviceIoControl.IoControlCode != IOCTL_SWITCH_STACKS)
    {
        Status = STATUS_INVALID_PARAMETER;
    }
    else
    {
        // Get the pointer to the function to call
        VOID (*UserFunctToCall)(PULONG) = Irp->UserBuffer;
        // And the argument to pass
        PVOID UserArg;
        UserArg = Ios->Parameters.DeviceIoControl.Type3InputBuffer;
        // Call user's function with the parameter
        (VOID)(*UserFunctToCall)((UserArg));
        Status = STATUS_SUCCESS;
    }
    Irp->IoStatus.Status = Status;
    Irp->IoStatus.Information = 0;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);
    return(Status);
}
The driver is called using the standard Win32 system service call, as shown below:

```c
DeviceIoControl (hDriver,
    (DWORD) IOCTL_SWITCH_STACKS,
    &UserData,
    sizeof(PVOID),
    &OriginalWinMain,
    sizeof(PVOID),
    &cbReturned)
```

This example is not designed to encourage you to write programs that run in kernel mode, of course. However, what the example does demonstrate is that when your driver is running, it is really just running in the context of an ordinary Win32 program, with all its variables, message queues, window handles, and the like. The only difference is that it’s running in Kernel Mode, on the Kernel stack.
Chapter 6

Summary

So there you have it. Understanding context can be a useful tool, and it can help you avoid some annoying problems. And, of course, it can let you write some pretty cool drivers. Let us hear from you if this idea has helped you out. Happy driver writing!